Guiding Component-Based Hardware/Software Co-Verification with Patterns *

Juncao Li and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207, USA
{juncao, xie}@cs.pdx.edu

Huaiyu Liu
Corporate Technology Group
Intel Corporation
Hillsboro, OR 97124, USA
huaiyu.liu@intel.com

Abstract

In component-based hardware/software co-verification, properties of an embedded system are established from properties of its hardware and software components. A major challenge in component-based co-verification is the property formulation problem: (1) what are the system properties to verify, (2) what are the component properties needed for verifying the system properties, and (3) what are the environment assumptions for establishing these properties. We present a pattern-guided approach to the property formulation problem. We develop an embedded architecture description language (EADL). A key feature of EADL is its support to specification of architectural patterns for embedded systems. Such patterns capture recurring system structures and, furthermore, templates for properties to verify on systems following these patterns and strategies for decomposing system properties into component properties. We have applies EADL in co-verification of medical sensor systems, which shows that architectural patterns have major potential in facilitating component-based co-verification.

1 Introduction

In today's embedded system design, the boundary between hardware and software has become increasingly blurred: hardware and software closely interact and functionalities often migrate across the boundary. This demands hardware/software co-verification. At the same time, embedded systems are increasingly component-based in order to attain modularity, functional reuse, and configuration flexibility. Component-based embedded systems for a given application domain often have commonalities in their architectures.

In [18], we have developed a component-based approach to hardware/software co-verification of embedded systems. In this approach, embedded systems are structured following a component model that unifies the concepts of hardware IPs [8] (i.e., hardware components) and software components [16]. In this model, verified temporal properties of hardware and software components are associated with the components. A special type of component, *bridge component*, is introduced, which inter-connects hardware and software components and bridges the hardware/software semantic gaps. Our approach to co-verification is a synergistic integration of bottom-up component verification and top-down system verification. Hardware and software components are verified as they are assembled bottom-up. Properties of a primitive component (i.e., a component that is not composed from other components) are directly verified while properties of a composite component are verified on its abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed via recursive decompositions into its components. The decompositions reuse components as possible. Verified properties of the reused components are reused in constructing the abstractions for verifying properties of the system.

A major challenge in component-based co-verification is the property formulation problem: (1) what are the system properties to verify, (2) what are the component properties needed for verifying the system properties, and (3) what are the environment assumptions necessary for establishing these properties. The problem may significantly hinder effectiveness of component-based co-verification. The increasing adoption of assertion-based verification (ABV) [10] alleviates this problem since designers are required to formulate the component properties as a component is designed. However, this problem persists since, in essence, it is due to lack of knowledge about possible environments of components, and it also plagues ABV although on a lesser extent. In addition, ABV requires major manual efforts in property formulation. Therefore, it is highly desired for heuristics that can reduce the property and assumption formulation efforts for embedded systems and composite components which follow commonly used architectures.

In this paper, we present a pattern-guided approach to the property formulation problem. We develop an embedded architectural description language (EADL) that sup-

*This research was supported by Semiconductor Research Corporation, Contract RID 1356.001.

ports rigorous architectural specification of embedded systems and provides complete language support for the unified component model. A key feature of EADL is its support to specification of architectural patterns for embedded systems (i.e., recurring patterns of embedded system architectures). EADL supports association of property templates and property decomposition strategies with embedded system architectural patterns. Property templates are formulated on component templates that abstract real components and component templates are specified as complete or partial component interfaces which provide the semantic information needed for specification of property templates. The property decomposition strategies will be represented in term of templates for component properties needed for verifying the pattern-level properties and the assume-guarantee relations among component properties. For a system following such an architectural pattern, its properties can be instantiated from the property templates and are decomposed under the guidance of the decomposition strategies which consider the properties of reusable components. We have applied our approach in component-based co-verification of medical sensor systems [15]. The case study has shown that architectural patterns have major potential in improving effectiveness of component-based co-verification by contributing to solution of the property formulation problem.

The reminder of this paper is organized as follows. In Section 2, we review the key concepts of component-based co-verification. In Section 3, we present EADL and how it captures architectural patterns. In Section 4, we elaborate on how architectural patterns are utilized in addressing the property formulation problem. In Section 5, we discuss the related work. In Section 6, we conclude this paper.

2 Component-Based Co-Verification

Our component-based co-verification approach [18] builds on and advances compositional model checking [3] by integrating compositional model checking into component-based development of embedded systems. It has three key features: (1) unified component model for hardware and software, (2) unified component property specification, and (3) integrated component and system verification.

2.1 Unified Component Model

Under the unified component model shown in Figure 1, an

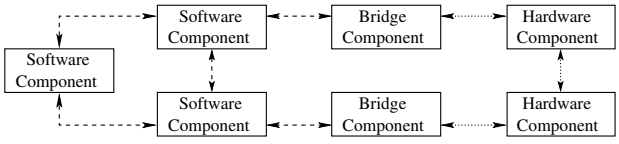

Figure 1. Unified Component Model

embedded system is assembled from components. There are three types of primitive components: *software components*, *hardware components*, and *bridge components*. Bridge components interact with hardware (or software, respectively) components following hardware (or software) semantics and bridge the semantic gap between hardware and software components by propagating events across the HW/SW semantic boundary. The semantics of bridge components together with the hardware and software semantics abstract the processor, bus model, and embedded OS of the targeted embedded system platform. Three types of composite components may also be defined: *software components*, *hardware components*, and *hybrid components*. Subcomponents of a composite software (or hardware, respectively) components are all software (or hardware) components. A hybrid component contains both hardware and software sub-components and, therefore, also bridge subcomponents. This model essentially integrates hardware and software component models.

Components. A component C is a triple (E, I, P) where E is the design or implementation of C, I is an interface including the semantic entities for C to interact with its environment and/or for specification of properties of C, and P is a set of temporal properties that are defined on I and have been verified on E. Hardware, software, and bridge components differ in the representations of E and I, but share the same representation of P. Each entry of P is a pair $(p, A(p))$ where p is a temporal assertion and $A(p)$ is a set of assumptions (i.e., assumed properties) on the environment of C for enabling the verification of p on C. The environment of C include components that interact with C in a system, and may be different in each system.

For a software component, E can be specified in C or other software design/programming languages. To support high-level design, we adopt the model-driven development [13] and specify software components in xUML [13], an executable dialect of UML. I of a software component is a pair, (M, V), where M is a set of input and output messages and V is a set of variables in E that are exported. The component communicates with its environment via asynchronous message-passing. The variables in V are to be mapped to hardware signals and/or to be utilized in specifying component properties and scheduling constraints. This interface semantics is determined by the asynchronous interleaving message-passing semantics of xUML.

For a hardware component, E can be specified in Verilog or other hardware design languages. In our study, we specify E in Verilog. I consists of a set of variables that the hardware component imports from or exports to its environment. The component communicates with its environment via variables in I. This interface semantics is determined by the synchronous clock-driven semantics of Verilog.

Bridge components inter-connect hardware and software components. The interface of a bridge component is a pair

(I_H, I_S). I_H is a synchronous shared-variable interface for interactions with hardware components and I_S is an asynchronous message-passing interface for interactions with software components. The interface of the bridge component is determined by the hardware and software components it connects. The design E of a bridge component is formulated in a domain-specific bridge spec language [18].

Composition. A composite component, $C = (E,\ I,\ P)$, is composed from a set of components, $C_0 = (E_0,\ I_0,\ P_0)$, ..., $C_{n-1} = (E_{n-1},\ I_{n-1}, P_{n-1})$, as follows. E is constructed from E_0, ..., E_{n-1} by connecting E_0, ..., E_{n-1} through I_0, ..., I_{n-1}. I may be a hardware interface, a software interface, or a hybrid hardware/software interface depending what types of components C_0, ..., C_{n-1} are. Essentially, I includes the semantic entities from I_0, ..., I_{n-1} that are needed for C to interact with its environment and/or for specification of properties of C. Properties of a composite component are established from properties of its sub-components (see Section 2.3).

2.2 Unified Property Specification

In [17], we develop a unified property specification language for HW/SW co-verification of embedded systems. This language, namely xPSL, builds on the IEEE Property Specification Language (PSL) [7]. It extends PSL to support specification of temporal assertions over both hardware and software events. The HW/SW semantic gap is filled by formalizing the semantics of hardware and software events and their temporal correlations based on translation of hardware and software semantics to a common formal semantic basis, in our case, the ω-automata semantics [9]. While PSL supports both LTL and CTL style temporal operators, xPSL inherits the linear-time subset of the PSL temporal operators and is, therefore, fully subsumed by ω-automata in expressiveness. xPSL is fully compatible with PSL and readily supports ABV. xPSL facilitates verification reuse: Properties of hardware and software components in xPSL can serve as abstractions of the components in system-level verification and can be reused across multiple systems if the components are reused. In this paper, we specify properties using a set of intuitive temporal templates based on xPSL and ω-automata. (See Section 3 for example properties.)

2.3 Integrated Component and System Verification

Under the component-based approach to co-verification, hardware and software components are verified as they are developed bottom-up. Properties of a primitive component are directly model-checked and properties of a composite component are checked on its abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed via recursive decompositions into its components. The decompositions reuse components as possible. Verified properties of the reused components are reused in constructing the abstractions for verifying properties of the system or higher-level components.

Given a property $(p, A(p))$ of a composite component C which is composed from C_0, ..., C_{n-1}, an abstraction of C for verifying $(p, A(p))$ is constructed as follows:

1. Construct a system of non-deterministic ω-automata ω_0, ..., ω_{n-1} each of which corresponds to C_i, $0 \leq i < n$, and simulates the interface of C_i. The ω-automata are inter-connected through the interfaces that they simulate, following the inter-connections among C_0, ..., C_{n-1}. A non-deterministic ω-automaton ω_e that simulates the environment of C is also added to ensure that the system is closed.

2. Constrain ω_i, $0 \leq i < n$, by composing ω_i with the ω-automata translated from the properties of C_i that are related to $(p, A(p))$ by cone-of-influence [3, 9] analysis and *enabled*. Constrain ω_e by composing ω_e with ω-automata translated from the assumptions in $A(p)$.

A property of C_i is *enabled* if and only if its assumptions are implied by the enabled properties of other sub-components and/or the assumptions in $A(p)$. There may exist circular dependencies among the sub-component properties. Suppose we have verified that a property, Q_0, holds on C_0 assuming that a property, Q_1, holds on C_1 and vice versa. We cannot conclude that Q_0 and Q_1 hold on C unless we can show that circular reasoning can be avoided. Circular reasoning can be avoided using the following methods (but not limited to these methods): (1) avoid using an assumption that creates a dependency cycle; (2) use temporal induction proposed by McMillan [11]; or (3) use the compositional reasoning rule proposed by Amla, et al. [2].

The abstraction constructed above is conservative. If the property holds on the abstraction, it also holds on the composite component; otherwise, the abstraction can be refined by verifying additional sub-component properties and including them in the abstraction. If the property does not hold on the composite component, abstraction refinement together with error trace analysis will uncover the cause.

3 Embedded Architecture Description Language

The unified component model defines the concept of component and basic rules for component composition. However, it is yet to support architectural specification, i.e., specification of an embedded system (or a composite component) in term of the inter-connecting relations among its hardware and software components (or sub-components). Architectural specifications are crucial to exploring compositional structures of embedded systems for co-verification.

We have developed an embedded architecture description language (EADL) that provides complete language support for the unified component model and for architectural specification of entire embedded systems. EADL extends the concept of software architecture [14] to the concept of embedded system architecture. A first desirable feature of EADL, in addition to the unified component representation above, is to support specification of the interconnecting relations among hardware, software, and bridge components in a system or a composite component. Another desirable feature of EADL is to support specification of architectural patterns of embedded systems, i.e., recurring patterns of embedded system architectures. We will discuss how architectural patterns can contribute to solution of the property formulation challenge in Section 4.

3.1 Embedded System Architecture Specification

To specify inter-connecting relations among hardware, software, and bridge components, EADL needs additional language constructs besides components and simple interfaces. We introduce events, ports, connectors, and configurations, and show how they are utilized in specifying architectures.

3.1.1 Events and Ports

We employ the event concept to abstract all concrete interaction mechanisms: messages, function calls, signals, etc. The event semantics are precisely defined when the EADL is instantiated for a specific embedded system platform. Events in an embedded systems can be of different semantics due to the differences between hardware and software semantics. This enables EADL to span across hardware and software semantics. Each event has an *in* or *out* direction. A port groups events into a clear identified functionality and it also has an *in* and/or *out* direction. Figure 2 shows the interface of a pulse sensor component, which consists of two ports, *StdCtrl* and *GSI*, where events are messages. The

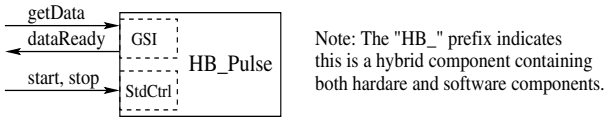

Figure 2. Interface of Pulse Sensor

EADL specification of the two ports is shown in Figure 4. Take *GSI* as an example, it groups all events necessary in the functionality of providing data to other components. The *in* direction indicates that the *GSI* port only responds to data requests and never provides data without a request.

3.1.2 Connectors and Configurations

To present a uniform and abstract view of embedded system architectures, we introduce the concept of connector. Connectors are components dedicated to connecting other components. Instead of connecting components on the detailed event level, connectors connect components on the more abstract port level. The ports in the interface of a connector match the ports in the interfaces of the components it connects: having the same events while the directions of the events are reversed. The implementation of the connector consists of the detailed mappings among the events from the ports in its interface. The EADL connector concept differs from the connector concept in software architectures in that the EADL connectors are extended to connect components of different semantics: hardware (or software) components of different semantics, e.g., hardware components in Verilog or VHDL, and furthermore hardware and software components. Essentially, it generalizes the bridge component concept in the unified component model. A connector has a direction based on the ports that it connects. The direction of a connector indicates which component can potentially initiate the interaction. It should be consistent with the directions of the ports involved. Figure 3 shows the architecture of the pulse sensor whose hardware and software sub-components are connected by a connector that bridges the hardware and software semantics. The software, hardware, and bridge sub-components are marked with the prefixes, *SW*, *HW*, and *BG*, respectively.

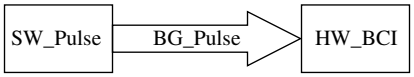

Figure 3. Architecture of Pulse Sensor

EADL specifies the architecture of a composite component through specifying its configuration which consists of all its sub-components, and the connectors among them. The configuration of the pulse sensor is shown in Figure 4. Besides the sub-components and connectors, configuration maps are also provided to indicate the correspondence between the ports/variables of the composite component and the ports/variables of its sub-components. Such a map can be one-to-one or one-to-many. For a primitive component, the configuration is replaced by its source code.

3.1.3 Properties

To support verification and reuse, EADL inherits from the unified component model the way how component properties are specified. The properties of a component, as its sub-components and its configuration, are part of the component specification and are defined over its ports and variables made visible for property specification. When a component is developed, its properties are established and can then be reused with the component. Environment assumptions for enabling verification of a component property are provided with the property. A property *SEN1* of the pulse sensor is shown in Figure 4, which asserts that after the pulse sensor

```
Component HB_Pulse {
 Port In StdCtrl {Input: start, stop;}
 Port In GSI {Input: getData; Output: dataReady(int var);}
 int drvSenVar = 0, devSenVar = 0;

 Configuration {
  Component SW_Pulse {
   Port In StdCtrl {Input: start, stop;}
   Port In GSI {Input: getData; Output: dataReady(int var);}
   Port Out Ctrl {Output: setPW, clrPW, enInt;}
   Port In Data {Input: data(int var);} int drvSenVar = 0;
   Properties{
    PSW1: Assert After(GSI.getData)
                Eventually(GSI.dataReady(var=drvSenVar))
          Assume After(GSI.getData) Never(GSI.getData)
                UnlessAfter(GSI.dataReady);
    PSW2: Assert IfEventuallyAlways(!Data.data + Data.data(var>T))
                EventuallyAlways(drvSenVar>T);
   }
  }

  Component HW_BCI {
   Port In HWCtrl {Input: sig_setPW, sig_clrPW, sig_enInt;}
   Port Out HWDT {Output: sig_int(int var);} int devSenVar = 0;
   Properties {
    BCI1: Assert IfEventuallyAlways(devSenVar>T)
          EventuallyAlways(!HWDT.sig_int+HWDT.sig_int(var>T));
   }
  }

  Connector BG_Pulse {
   Cnn(SW_Pulse.Ctrl.setPW, HW_BCI.HWCtrl.sig_setPW);
   Cnn(SW_Pulse.Ctrl.clrPW, HW_BCI.HWCtrl.sig_clrPW);
   Cnn(SW_Pulse.Ctrl.enInt, HW_BCI.HWCtrl.sig_enInt);
   Cnn(SW_Pulse.Data.data, HW_BCI.HWDT.sig_int);
  }

  ConfigMap(SW_Pulse.StdCtrl, StdCtrl);
  ConfigMap(SW_Pulse.GSI, GSI);
  ConfigMap(SW_Pulse.drvSenVar, drvSenVar);
  ConfigMap(HW_BCI.devSenVar, devSenVar);
 }

 Properties {
  SEN1: Assert After(GSI.getData)
              Eventually(GSI.dataReady(var=drvSenVar))
        Assume After(GSI.getData) Never(GSI.getData)
              UnlessAfter(GSI.dataReady);
  SEN2: Assert IfEventuallyAlways(devSenVar>T)
              EventuallyAlways(drvSenVar>T);
 }
}
```

Figure 4. EADL Spec for Pulse Sensor

receives a *getData* message, it will eventually reply with a *dataReady* message whose data field equals to *drvSenVar*, assuming that no further *getData* message is received unless after the previous *getData* message has been acknowledged.

3.2 Embedded System Architectural Patterns

With the above features of EADL, we can specify the architectures of embedded systems and their components. There often exist common patterns among the architectures of systems or components, e.g., the CodeBlue architectural pattern [15] for medical sensor systems in Figure 5. Furthermore, sensor components used in the CodeBlue pattern often share the same architectural pattern as the pulse sensor.

To capture such a pattern, we need additional language constructs. While the architecture of a system or component is captured as a configuration which is based on composition of components, an architectural pattern is captured as a configuration template which is based on composition of component templates. Abstraction of patterns from the component/system architectures is based on abstraction of component templates from components. A component template is a skeleton for components, which captures the parameterized interface shared by these components, the common set of variables of these components, and the templates for properties of these components. The property templates are defined over the parameterized interface and the variable set in the component template. As the component template is instantiated into a component, the property templates are instantiated into component properties. Take the CodeBlue pattern as an example, the software component *CodeBlueQuery (CBQ)* manages different numbers and kinds of sensors in different systems, delivers a query request to the right sensor, and filters data with thresholds before output. Because most CodeBlue system instances are differentiated by the specific sensors and their associated data filters, the *CBQ* component which deals with sensors directly can be abstracted as a component template *T_SW_CBQ* to support the abstraction of the CodeBlue pattern. Shown in Figure 6 is the specification of this template, which is composed of port templates, variables, and property templates. As the

```
Component Template T_SW_CBQ {
 Port In StdCtrl {Input: start, stop;}
 Port In QueryHandler {Input: handleQuery(int src, sink, threshold),
                              cancelQuery(int src, sink);
                       Output: dataReady(int var); }
 Multi Sen[NumofSen] {
  Port Out Ctrl {Output: start, stop;}
  Port Out Data {Output: getData; Input: dataReady(int var);}
 }
 int QrySink[NumofSen], QryThreshold[NumofSen];

 Properties{
  CBQ1: Assert After(QueryHandler.handleQuery(SRC, SINK, T))
              Repeatedly(Sen[SRC].Data.getData)
              UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
  CBQ2: Assert After(QueryHandler.handleQuery(SRC, SINK, T))
              EventuallyAlways ((QrySink[SRC]=SINK)
                                *(QryThreshold[SRC]=T))
              UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
  CBQ3: Assert After((Sen[SRC].Data.dataReady(var>QryThreshold[SRC]))
                     *(QrySink[SRC]=SINK))
              Eventually(QueryHandler.dataReady(SRC, SINK,
                                (var>QryThreshold[SRC])))
              UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
  CBQ4: Assert After(Sen[SRC].Data.getData) Never(Sen[SRC].Data.getData)
              UnlessAfter(Sen[SRC].Data.dataReady);
 }
}
```

Figure 6. EADL Spec for CBQ Template

number of sensors that *CBQ* manages varies depending on the individual system, the number of corresponding ports varies accordingly. Keyword *Multi* is used to support this feature. By using *Multi* to group the ports, it indicates this is a group of ports that can be instantiated multiple times according to the number of sensors in a specific system.

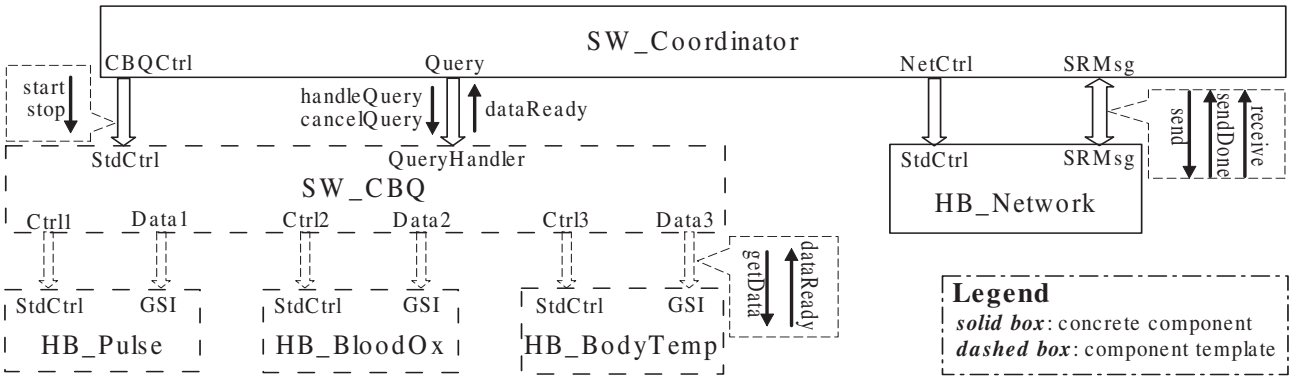

Figure 5. CodeBlue Architectural Pattern for Medical Sensor Systems

While the component level abstraction is achieved by component templates, the architecture level abstraction is realized by configuration templates. A configuration template consists of component templates, connector templates, and also concrete components and connectors. Shown in Figure 7 is the specification of the CodeBlue architectural pattern, in which two concrete components and two component templates are included. In particular, *T_HB_Sensor*

```
Pattern P_HB_CodeBlue {
  Configuration Template CodeBlueCnfg{
    Component SW_Coordinator;
    Component HB_Network;
    Component Template T_SW_CBQ;
    Multi Component Template T_HB_Sensor[NumofSen];
    Connector SW_CrdNetCnn{
      Cnn(SW_Coordinator.NetCtrl, HB_Network.StdCtrl);
      Cnn(SW_Coordinator.SRMsg, HB_Network.SRMsg);}
    Connector SW_CrdCBQCnn{
      Cnn(SW_Coordinator.CBQCtrl, T_SW_CBQ.StdCtrl);
      Cnn(SW_Coordinator.Query, T_SW_CBQ.QueryHandler);}
    Multi Connector Template T_SW_CBQSenCnn{
      Cnn(T_SW_CBQ.Ctrl, T_HB_Sensor.StdCtrl);
      Cnn(T_SW_CBQ.Data, T_HB_Sensor.GSI);}
  }

  Properties{
   CB1: Assert After(SW_Coordinator.Query.handleQuery(SRC, SINK, T))
                Eventually(SW_Coordinator.SRMsg.send(SRC, SINK, var>T))
                UnlessAfter(SW_Coordinator.Query.cancelQuery(SRC, SINK))
        Assume After(SW_Coordinator.Query.handleQuery(SRC, SINK, T))
                EventuallyAlways(T_HB_Sensor[SRC].devSenVar>T);
  }
}
```

Figure 7. EADL Spec for CodeBlue Pattern

is a *Multi* component template that is mapped to the *Sen* interface of the *T_SW_CBQ* component template. It can be instantiated for multiple times in a system instance.

Based on the above abstractions, an architectural pattern consists of three parts similar to those of a component architecture: (1) an interface template for a component/system following this pattern, which consist of port templates and variables; (2) a configuration template; (3) templates for properties of a component/system following this pattern. (Note that the CodeBlue Pattern has no interface template.)

4 Pattern-Guided Co-Verification

As discussed in the previous section, EADL not only provides the language support for capturing architectures and architectural patterns of embedded systems, but also supports association of properties with components and properties templates with architectural patterns. Our approach utilizes EADL to address the property formulation challenge in the following ways: (1) pattern-guided property formulation, (2) pattern-guided property decomposition, and (3) pattern-guided circular reasoning prevention.

4.1 Pattern-Guided Property Formulation

Patterns can guide property formulation for both systems and reusable components. A difficulty in ABV is to identify the potential environments for a reusable component and how it interacts with the environments. Patterns essentially abstract the potential environments for reusable components. If a component is designed to be reused under a given pattern, the pattern often determines the properties that ought to hold on the component and the appropriate environment assumptions of these properties. An example is the *T_SW_CBQ* component template under the CodeBlue pattern which dictates the interactions between *CBQ* and other components and suggests the properties in Figure 6.

Another difficulty in property formulation is how to derive appropriate behavior rules of the system from the system requirements, i.e. what are the system properties to verify. Patterns are utilized as the vehicle to address this problem during the system design process. Given the system requirements, the architecture patterns used to structure the system are selected. These patterns suggest what properties to verify on the system. In EADL, this is supported by pattern-level property templates which are specified on the system-level (or composite component level) interfaces and variables. Property *CB1* in Figure 7 is such an example. It asserts that for a medical sensor system following the CodeBlue pattern, if there is an active query on a given sensor

and the sensor's reading exceeds the corresponding threshold, the system will eventually report the sensor's reading.

4.2 Pattern-Guided Property Decomposition

In component-based development of embedded systems, the system synthesis process is top-down, which recursively decomposes a system into its components and their interconnecting relations, until reaching the primitive components or the reusable components from the library. The synthesis process often depends on knowledges and experiences of system architects to determine the decompositions, e.g., hardware and software partitions. Architectural patterns facilitate this process through capturing reusable knowledges about system architectures.

The pattern-guided synthesis process shown in Figure 8 starts with pattern selection. In this step, architectural patterns are manually selected according to the specification of system/component requirement and interface. If a pattern is successfully selected from the pattern library, the system/component is decomposed into concrete components and component templates following the pattern. The decomposition stops at a concrete component which is reused directly. A component template, which is only a component skeleton with ports and property templates, is utilized to select a component matching the template specification. This decomposition stops if the component can be reused; otherwise, there are two ways to design the component: as a primitive component by which the decomposition stops with direct implementation or as a composite component by which the decomposition continues recursively. If there is no appropriate pattern that matches the system/component specification, the decomposition is conducted manually.

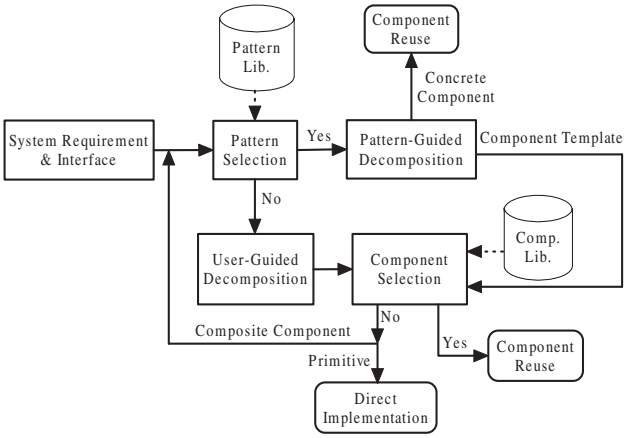

Figure 8. Top-Down System Decomposition

To enable efficient component-based co-verification, we integrate verification into the top-down synthesis process, in particular, integrating property decomposition with system decomposition, which is a major advantage of our approach. As an architectural pattern is selected to guide a decomposition, its associated property decomposition strategy is also utilized to decompose the pattern-level properties into the component properties. As a component template is utilized to select a component, both the interface/port templates and the property templates are used to guide the component selection. A basic approach to specification of decomposition strategies is to associate appropriate property templates with the component templates in a pattern and define the dependency links from a pattern-level property templates to the component property templates. Dependencies among component properties are captured as assumptions of these properties. For instance, for the pattern-level property of the CodeBlue pattern, we can define a decomposition strategy as shown in Figure 9. This strategy must be instantiated for each individual system since the properties of the *CBQ* component depends on how many sensors are included in the system.

```
// Pattern-level property
CB1: Assert After (SW_Coordinator.Query.handleQuery(SRC, SINK, T))
              Eventually (SW_Coordinator.SRMsg.send(SRC, SINK, var>T))
              UnlessAfter (SW_Coordinator.Query.cancelQuery(SRC, SINK))
       Assume After (SW_Coordinator.Query.handleQuery(SRC, SINK, T))
              EventuallyAlways (T_HB_Sensor[SRC].devSenVar>T);

// Properties of Coordinator
CRD1: Assert After (SW_Coordinator.Query.dataReady(SRC, SINK, var>T))
              Eventually(SW_Coordinator.SRMsg.send(SRC, SINK, var>T));
CRD2: Assert Never (SW_Coordinator.Query.handleQuery(SRC, SINK, T)*
                    SW_Coordinator.Query.cancelQuery(SRC, SINK));

// Properties of CBQ as shown in Figure 6

// Properties of Sensors as shown in Figure 4

// Decomposition strategy
CB1 → CRD1, CRD2, CBQ1, CBQ2, CBQ3, CBQ4, SEN1, SEN2;
```

Figure 9. An example decomposition strategy

This straightforward approach to strategy specification is cumbersome when a lot of property templates need to be specified. And as discussed above, for certain patterns the number of components that are involved in the patterns are only known when the components are instantiated. Therefore, more convenient ways to specify decomposition strategies is needed. We are currently working on providing language supports for specifying decomposition strategies as simple programs. These programs need to consider other knowledge that is useful in property decomposition, e.g., certain port templates and connector templates may dictate the way how component properties must be formulated.

4.3 Pattern-Guided Circular Reasoning Avoidance

There may exist assume-guarantee dependency cycles among component properties, which can potentially lead to circular reasoning. Property dependency cycles may be accidentally introduced by incorrect formulation of compo-

nent properties or intentionally introduced to reflect the nature of component interactions and simplify property specification. The first type of cycles can be eliminated by cycle detection. For the second type of cycles, additional work may be needed to show that such cycles will not cause circular reasoning, for instance, using the methods discussed in Section 2.3. Solution of this problem can be made more efficient through architectural patterns since once a pattern can be shown free of circular reasoning, the instantiations of the pattern are free of circular reasoning. In essence, the circular reasoning detection is conducted only once for the pattern and is reused when the pattern is reused.

4.4 Preliminary Evaluation

We have applied EADL and the pattern-guided approach to the medical sensor systems in the CodeBlue software package [15]. We started by capturing the CodeBlue pattern using EADL. We then decomposed the Pulse Oximeter Sensor system, one of the systems in the package, formulated the properties of the system and its components, and conducted compositional model checking. We then abstracted the system property template from the system property and the property templates from the component properties and used these templates to guide verification of two other systems in the package. This case study shows our approach has major potential in verifying embedded system families.

5 Related Work

Pattern reuse is often conducted at two levels: design and architecture level. Design patterns [4] are concerned with reuse of programming structures at the algorithmic or data structure level. Architectural patterns [14] are concerned with reusable structural patterns of software system with respect to their components. Architectural patterns have been applied in software design, documentation, validation, etc. Our research utilizes architectural patterns to facilitate formulation of properties of embedded systems and their components. There exist many software architectural description languages [12]. Among them, ACME [5] is closely related to our work. It provides language constructs for specifying architectural patterns. Our representation of architectural patterns is partially motivated by that of ACME and specially targets HW/SW co-design and co-verification of component-based embedded system families.

There are also approaches [1, 6] to automatic generation of assumptions for safety properties of components. Our approach addresses the property formulation challenge via architectural pattern guided property formulation in bottom-up component verification and via pattern guided property decomposition in top-down system verification. It handles both safety and liveness properties and complements automatic assumption generation for safety properties.

6 Conclusions and Future Work

We have presented an approach to guiding component-based hardware/software co-verification with architectural patterns. Architectural patterns are extended to capture templates for properties to be verified on systems following these patterns and strategies to decompose these properties. Future work will be focused on further automation of our pattern-guided approach in formulation of property patterns and decomposition strategies and instantiations of property patterns for specific systems. We will also apply this approach to a broader range of embedded systems families.

References

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, 2005.

[2] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *Proc. of TACAS*, 2001.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software*. Addison-Wesley, 1994.

[5] D. Garlan, R. T. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON*, 1997.

[6] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, 2002.

[7] IEEE. *IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2005)*. IEEE, 2005.

[8] M. F. Jacome and H. P. Peixoto. A survey of digital design reuse. *IEEE Design and Test of Computers*, 18(3), 2001.

[9] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[10] D. Maliniak. Assertion-based verification smooths the road to IP reuse. *Electronic Design*, September 2002.

[11] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence Design Systems Technical Reports*, 1999.

[12] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1), 2000.

[13] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.

[14] M. Shaw and D. Garlan. *Software Architecture: Perspective on An Emerging Discipline*. Prentice Hall, 1996.

[15] V. Shnayder, B. R. Chen, K. Lorincz, T. R. F. Fulford-Jones, and M. Welsh. Sensor networks for medical care,. Technical report, Harvard University, 2005.

[16] C. Szyperski and et al. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 2002.

[17] F. Xie and H. Liu. Unified property specification for hardware/software co-verification. Proc. of COMPSAC, 2007.

[18] F. Xie, G. Yang, and X. Song. Component-based hardware/software co-verification for building trustworthy embedded systems. *Journal of Systems and Software*, 2007.